A Generic Software Safety Document Generator

Ewen Denney[1] and Ram Prasad Venkatesan*[,2]

[1] QSS Group Inc, NASA Ames Research Center, Moffett Field, CA
edenney@email.arc.nasa.gov
[2] Department of Computer Science, University of Illinois at Urbana-Champaign, IL
rpvenkat@uiuc.edu

Abstract. *Formal certification* is based on the idea that a mathematical proof of some property of a piece of software can be regarded as a certificate of correctness which, in principle, can be subjected to external scrutiny. In practice, however, proofs themselves are unlikely to be of much interest to engineers. Nevertheless, it is possible to use the information obtained from a mathematical analysis of software to produce a detailed textual justification of correctness. In this paper, we describe an approach to generating textual explanations from automatically generated proofs of program *safety*, where the proofs are of compliance with an explicit *safety policy* that can be varied. Key to this is tracing proof obligations back to the program, and we describe a tool which implements this to certify code auto-generated by AutoBayes and AutoFilter, program synthesis systems under development at the NASA Ames Research Center. Our approach is a step towards combining formal certification with traditional certification methods.

1 Introduction

Formal methods are becoming potentially more applicable due, in large part, to improvements in automation: in particular, in automated theorem proving. However, this increasing use of theorem provers in both software and hardware verification also presents a problem for the applicability of formal methods: how can such specialized tools be combined with traditional process-oriented development methods?

The aim of formal certification is to prove that a piece of software is free of certain defects. Yet certification traditionally requires documentary evidence that the software development complies with some process (e.g., DO-178B). Although theorem provers typically generate a large amount of material in the form of formal mathematical proofs, this cannot be easily understood by people inexperienced with the specialized formalism of the tool being used. Consequently, the massive amounts of material that experts can create with these theorem provers fairly inaccessible. If you trust a theorem prover, then a proof of correctness tells that a program is safe, but this is not much help if you want to understand *why*.

* Ram Prasad Venkatesan carried out this work during a QSS summer internship at the NASA Ames Research Center.

One approach is to verbalize high-level proofs produced by a theorem prover. Most of the previous work in this direction has focused on translating low-level formal languages based on natural deduction style formal proofs. A few theorem provers, like Nuprl [CAB+86] and Coq [BBC+97], can display formal proofs in a natural language format, although even these readable texts can be difficult to understand. However, the basic problem is that such proofs of correctness are essentially stand-alone artifacts with no clear relation to the program being verified.

In this paper, we describe a framework for generating comprehensive explanations for *why* a program is safe. Safety is defined in terms of compliance with an explicitly given safety policy. Our framework is generic in the sense that we can instantiate the system with a range of different safety policies, and can easily add new policies to the system.

The safety explanations are generated from the proof obligations produced by a verification condition generator (VCG). The verification condition generator takes as input a synthesized program with logical annotations and produces a series of verification conditions. These conditions are preprocessed by a rewrite-based simplifier and are then proved by an automated theorem prover. Unfortunately, any attempt to directly verbalize the proof steps of the theorem prover would be ineffective as

- the process of simplifying the proof objects makes it difficult to provide a faithful reproduction of the entire proof;
- it is difficult to relate the simplified proof obligations to the corresponding parts of the program.

We claim that it is unnecessary to display actual proof steps — the proof obligations alone provide sufficient insight into the safety of a program. Hence we adopt an approach that generates explanations directly from the verification conditions. Our goals in this paper are:

- using natural language as a basis for safety reports;
- describing a framework in which proofs of safety explicitly refer back to program components;
- providing an approach to merge automated certification with traditional certification procedures.

Related Work Most of the previous work on proof documentation has focused on translating low-level formal proofs, in particular those given in natural deduction style. In [CKT95], the authors present an approach that uses a proof assistant to construct proof objects and then generate explanations in pseudo-natural language from these proof objects. However, this approach is based on a low-level proof even when a corresponding high-level proof was available. The Proverb system [Hua94] renders machine-found natural deduction proofs in natural language using a reconstructive approach. It first defines an intermediate representation called *assertion level* inference rules, then abstracts the machine-found natural deduction proofs using these rules; these abstracted proofs are

then verbalized into natural language. Such an approach allows atomic justifications at a higher level of abstraction. In [HMBC99], the authors propose a new approach to text generation from formal proofs exploiting the high-level interactive features of a tactic-style theorem prover. It is argued that tactic steps correspond approximately to human inference steps. None of these techniques, though, is directly concerned with program verification. Recently, there has also been research on providing formal traceability between specifications and generated code. [BRLP98] presents a tool that indicates how statements in synthesized code relate to the initial problem specification and domain theory. In [WBS+01], the authors build on this to present a documentation generator and XML-based browser interface that generates an explanation for every executable statement in the synthesized program. It takes augmented proof structures and abstracts them to provide explanations of how the program has been synthesized from a specification.

One tool which does combine verification and documentation is the PolySpace static analysis tool [Pol]. PolySpace analyzes programs for compliance with fixed notions of safety, and produces a marked-up browsable program together with a safety report as an Excel spreadsheet.

2 Certification Architecture

The certification tool is built on top of two program synthesis systems. AutoBayes [FS03] and AutoFilter [WS03] are able to auto-generate executable code in the domains of data analysis and state estimation, respectively. Both systems are able to generate substantial complex programs which would be difficult and time-consuming to develop manually. Since these programs can be used in safety-critical environments, we need to have some guarantee of correctness. However, due to the complex and dynamic nature of the synthesis tools, we have departed from the traditional idea of program synthesis as being "correct by construction" or *process-oriented certification*, and instead adopt a *product-oriented* approach. In other words, we certify the individual programs which are generated by the system, rather than the system itself.

Figure 1 gives an overview of the components of the system. The synthesis system takes as input a high-level specification together with a *safety policy*. Low-level code is then synthesized to implement the specification. The synthesizer first generates "intermediate" code which can then be translated to different platforms. A number of target language backends are currently supported. The safety policy is used to *annotate* the intermediate code with mark-up information relevant to the policy. These annotations give "local" information, which must then be propagated throughout the code. Next, the annotated code is processed by a Verification Condition Generator (VCG), which applies the rules of the safety policy to the annotated code in order to generate safety conditions (which express whether the code is safe or not). The VCG has been designed to be "correct-by-inspection", that is, sufficiently simple that it is relatively easy to be assured that it correctly implements the rules of the safety logic. In particular,

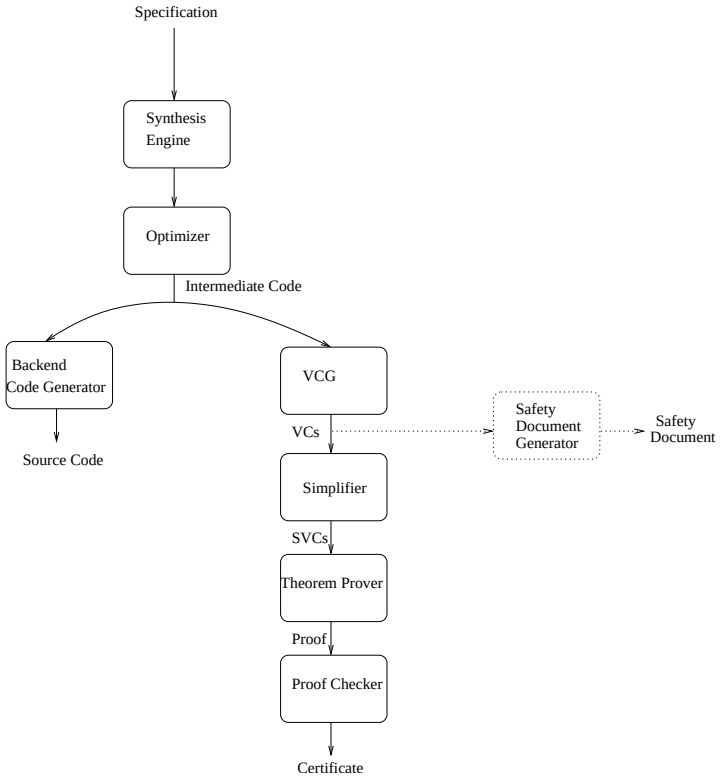

Fig. 1. Certification Architecture

the VCG does not carry out any optimizations, not even reducing substitution terms. Consequently, the verification conditions (VCs) tend to be large and must be preprocessed before being sent to a theorem prover. The preprocessing is done by a traceable rewrite system. The more manageable SVCs are then sent to a first-order theorem prover, and the resulting proof is sent to a proof checker. In the above diagram, the safety documentation extension is indicated using dotted lines.

3 Safety Policies

Formal reasoning techniques can be used to show that programs satisfy certain *safety policies*, for example, memory safety (i.e. they do not access out of bound memory locations), and initialization safety (i.e. uninitialized variables are not used). Formally, a safety policy is a set of proof rules and auxiliary definitions which are designed to show that programs satisfy a safety property of interest. The intention is that a safety policy enforces a particular safety property, which

is an operational characterization that *a program does not go wrong*. The distinction between safety properties and policies is explored in detail in [DF03]. We summarize the important points here.

Axiomatic semantics for (simple) programming languages are traditionally given using Hoare logic [Mit96], where $P\ \{C\}\ Q$ means that if precondition, P, holds before the execution of command, C, then postcondition, Q, holds afterwards. This can be read backwards to compute the weakest precondition which must hold to satisfy a given postcondition.

We have extended the standard Hoare framework with the notion of safety properties. [DF03] outlines criteria when a (semantic) safety property can be encoded as an executable safety policy.

Hoare logic treats commands as transformations of the execution environment. The key step in formalizing safety policies is to extend this with a "shadow", or safety environment. Each variable (both scalar and vector) has a corresponding shadow variable which records the appropriate safety information for that variable. For example, for initialization safety, the shadow variable x_{init} is set to *init* or *uninit* depending on whether x has been initialized or not. In general, there is no connection between the values of a variable and its shadow variables. The semantic definition of a safety property can then be factored into two families of formulas, $\mathsf{Safe}^{\_}_{\_}$ and $\mathsf{Sub}^{\_}_{\_}(\_)$. A feature of our framework is that the safety of a command can only be expressed in terms of its *immediate* subexpressions. The subscripts give the class of command (assignment, for-loop, etc.), and the superscript lists the immediate subexpressions.

For a given safety policy, for each command, C, of class `cl` with immediate subexpressions, $e_1 \ldots e_n$, $\mathsf{Safe}^{e_1 \cdots e_n}_{\mathtt{cl}}$ expresses the safety conditions on C, in terms of program variables and shadow variables; $\mathsf{Sub}^{e_1 \cdots e_n}_{\mathtt{cl}}(P)$ is a substitution applied to formula P expressing the change C makes to the shadow environment.

For example, for the initialization safety policy, the assignment $x := y$ has safety condition, $\mathsf{Safe}^{x,y}_{\mathtt{assign}}$, which is the formula $y_{\text{init}} = \mathit{init}$ (i.e., "y must be initialized") and, for formula P, $\mathsf{Sub}^{x,y}_{\mathtt{assign}}(P)$ is the substitution $P[\mathit{init}/x]$ (i.e., "x becomes initialized").

Hence, in our framework, *verifying* the safety of a program amounts to working backwards through the code, applying safety substitutions to compute the safety environment, and accumulating safety obligations while proving that the safety environment at each point implies the corresponding safety obligations. *Explaining* the safety of a program amounts to giving a textual account of why these implications hold, in terms relating to the safety conditions and safety substitutions.

Our goal, then, is to augment the certification system such that the proof obligations have sufficient information that we can give them a comprehensible textual rendering. We do this by extending the intermediate code to accommodate labels and the VCG to generate verification conditions with labels. We add labels for each declaration, assignment, loop construct and conditional statement by giving them a number in increasing order starting from zero. For loops and conditions, we also add the command type to the label. For example, `for` loops

are given a label `for`*(label)*. Similarly, we also have labels `if`(*label*) and `wh`(*label*). Figure 2 gives the Hoare rules extended by labels which are implemented by the VCG.

$$\text{(decl)}\quad \frac{}{\mathtt{lab}(l, \mathsf{Sub}^{x}_{\mathtt{decl}}(Q) \wedge \mathsf{Safe}^{x}_{\mathtt{decl}})\ \{(\mathtt{var}\ x)^l\}\ Q}$$

$$\text{(adecl)}\quad \frac{}{\mathtt{lab}(l, \mathsf{Sub}^{x,n}_{\mathtt{adecl}}(Q) \wedge \mathsf{Safe}^{x,n}_{\mathtt{decl}})\ \{(\mathtt{var}\, x\,\mathtt{[}n\mathtt{]})^l\}\ Q}$$

$$\text{(assign)}\quad \frac{}{\mathtt{lab}(l, \mathsf{Sub}^{x,e}_{\mathtt{assign}}(Q) \wedge \mathsf{Safe}^{x,e}_{\mathtt{assign}})\ \{(x\ \mathtt{:=}\ e)^l\}\ Q}$$

$$\text{(update)}\quad \frac{}{\mathtt{lab}(l, \mathsf{Sub}^{x,e_1,e_2}_{\mathtt{update}}(Q) \wedge \mathsf{Safe}^{x,e_1,e_2}_{\mathtt{update}})\ \{(x\mathtt{[}e_1\mathtt{]}\ \mathtt{:=}\ e_2)^l\}\ Q}$$

$$\text{(if)}\quad \frac{b \wedge P\ \{c_1\}\ Q \quad \neg b \wedge P\ \{c_2\}\ Q}{\mathtt{lab}(\mathtt{if}(l), \mathsf{Sub}^{b}_{\mathtt{if}}(P) \wedge \mathsf{Safe}^{b}_{\mathtt{if}})\ \{(\mathtt{if}\, b\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2)^l\}\ Q}$$

$$\text{(while)}\quad \frac{P\ \{c\}\ I \quad I \& b \Rightarrow P \quad I \& \neg b \Rightarrow Q}{\mathtt{lab}(\mathtt{wh}(l), \mathtt{inv}(I), \mathsf{Sub}^{b}_{\mathtt{while}}(I) \wedge \mathsf{Safe}^{b}_{\mathtt{while}})\ \{(\mathtt{while}\, b\, \mathtt{inv}\, I\, \mathtt{do}\, c)^l\}\ Q}$$

Fig. 2. Extended Hoare Rules

We have initially restricted ourselves to safety of array accesses (ensuring that the access is within the array bounds) and safety of variables with respect to initialization. However, we intend to extend our tool to support safety with respect to memory reads and writes, unit safety and data flow safety. This would be easily incorporated given the generic nature of our framework.

4 Documentation Architecture

In this section, we introduce the general architecture of the safety document generator and discuss the notions of def-use analysis and template composition.

4.1 Document Generator

Figure 3 shows the structure of the document generation subsystem. The synthesized intermediate code is labeled by adding line numbers to the code before it is sent to the VCG. The VCG then produces verification conditions for the corresponding safety policy. These verification conditions preserve the labels by encapsulating them along with the weakest safety preconditions.

The document generator takes as input the verification conditions generated in this manner and first extracts the needed information (more details are given in Section 5). It next identifies each part of the program that requires explanation and selects appropriate *explanation templates* from a repository of safety-dependent templates.

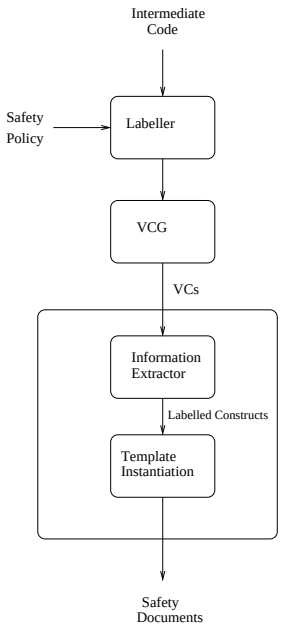

Fig. 3. Document Generation Architecture

Because of the way our safety logic is defined in terms of immediate subexpressions of commands, we define a *fragment* to be a command "sliced" to its immediate subexpressions. For atomic commands, this is equivalent to the command itself. For compound commands, we will represent this as `if` b and `while` b. These are the parts of a program that require an independent safety explanation. Text is then generated by instantiating the templates with program fragments.

4.2 Def-Use Analysis

Since commands can affect the safety of other commands in their effect on the program environment we cannot consider the safety of commands in isolation. In particular, the safety of a command involving a variable `x` depends on the safety of all previous commands in the program that contain an occurrence of `x`. Consider the following code:

```
    ⋮
(L1) x = 2
    ⋮
(L2) y = x
    ⋮
(L3) a[y] = 0
```

Now consider the safety of the expression `a[y] = 0` with respect to array bounds. To determine whether this access is safe, we need to ensure that the value held by `y` is within the array bounds. Now supposing that `a` is an array of size 10, we need to reason that `y` is defined from `x` which in turn is initialized to 2, which is less than 10. Hence we can state that the access is safe. Similarly, if we were analyzing the safety of the same expression with respect to initialization, we would need to convince ourselves simply that `y` is initialized. Reasoning from `y = x` alone would be insufficient and incorrect because `x` could be uninitialized. So we need to convince ourselves that `x` is also initialized by considering the expression `x = 2`. In other words, the safety of the expression `a[y] = 0` depends on the safety of the fragments `y=x` and `x=2`.

To summarize, we trace each variable in a program fragment ϕ to its origin (the point where it was first defined or initialized) and reason about the safety of all the fragments encountered in the path up to the origin to obtain a thorough explanation of the safety of ϕ. For a given program fragment ϕ having variables ω, we use $\Omega(\phi)$ to represent the set of all fragments, with their labels, that were encountered while tracking each variable in ω to its origin. We also include ϕ in $\Omega(\phi)$. Strictly speaking, the argument to Ω should be a distinguished occurrence of a fragment within a program, but we will gloss over this.

4.3 Contexts

In addition to tracking variables to their origin, we also need to find which fragments the fragment under consideration depends on. For example, the safety of an assignment statement appearing within a conditional block also depends on the safety of the conditional expression. Similarly, the safety of statements inside `while` loops depends on the safety of the loop condition. In the case of nested loops and conditional statements, a fragment's safety depends on multiple fragments. To provide complete safety explanations for a program fragment ϕ, we construct a set $\Psi_{sp}(\phi)$ as follows. As above, ϕ is assumed to be distinguished within a given program. We first identify all the fragments ϕ' on which ϕ depends. That is, if ϕ lies within conditional blocks and/or loop blocks, then we include the fragments representing those conditional expressions and/or loop expressions in ϕ'. We will refer to this as the *context* of the fragment, ϕ, and denote it by $\texttt{cxt}(\phi)$. Since we add special labels to loops and conditional statements, we can easily identify blocks. Hence even if a fragment ϕ is buried deep within conditions and nested loops, we can determine the set ϕ' with ease. Then, we trace each component and variable in the fragment ϕ and the set of fragments ϕ' to their origin (as explained in the previous section); that is,

$$\Psi_{sp}(\phi) = \cup\{\Omega(\phi') \mid \phi' \in \texttt{cxt}(\phi)\}.$$

Intuitively, we can view $\Psi_{sp}(\phi)$ as the set of all expressions and program fragments that we need to consider while reasoning about the safety of ϕ with respect to the safety policy *sp*. Each element in this set is represented as a (*label*, *fragment*) pair.

We now state (without proof) that ϕ is safe if each of the fragments in $\Psi_{sp}(\phi)$ is safe. That is,

$$safe_{sp}(\Psi_{sp}(\phi)) \Rightarrow safe_{sp}(\phi).$$

Here, we use the predicate *safe* to indicate that a set of program fragments are safe with respect to a policy *sp*.

For example, consider the following piece of code in C:

```
(1) x = 5 ;
(2) z = 10
(3) if(x > z)
(4)    y = x ;
    else
(5)    y = z ;
```

Here, the safety of the assignment `y = x` at line 4 with respect to initialization of variables depends not only on the assignment statement `y = x` but also on the the conditional fragment `if(x > z)` so, in this case, for the program fragment `y = x`, the context would be simply $\{\texttt{if(x} > \texttt{z)}\}$. We can further deduce that the safety of the conditional statement in turn depends on the two assignment statements `x = 5` and `z = 10`. So, to explain the safety of the expression `y = x` at line 4, we need to reason about the safety of the fragments `if(x > z)` , `z = 10` and `x = 5` at lines 3, 2 and 1 respectively. Hence, $\Psi_{sp}(\texttt{y} = \texttt{x})$ is the set $\{(4, \texttt{y} = \texttt{x}), (3, \texttt{if(x} > \texttt{z)}), (2, \texttt{z} = \texttt{10}), (1, (\texttt{x} = \texttt{5}))\}$.

4.4 Templates

We have defined a library of templates which are explanation fragments for the different safety policies. These templates are simply strings with holes which can be instantiated by program components to form safety explanations. A program component can be a simple program variable, a program fragment, an expression or a label.

Template Composition and Instantiation: The composition of an explanation for a given program fragment is obtained from the templates defined for a given policy, *sp*. For each fragment, ϕ, we first construct the set $\Psi_{sp}(\phi)$. Then, for each element ψ in $\Psi_{sp}(\phi)$, we find the required template(s), $Temp_{sp}(\psi)$. Next we insert the appropriate program components in the gaps present in the template to form the textual safety explanation. This process is repeated recursively for each fragment in $\Psi_{sp}(\phi)$ and then all the explanations obtained in this way are concatenated to form the final safety explanation. It should be noted that the safety explanations generated for most of these fragments are phrases rather than complete sentences. These phrases are then combined in such a manner that the final safety explanations reflects the data flow of the program.

As an example, consider the following code fragment:

```
(1) var a[10] ;
(2) x = 0 ;
(3) a[x] = 0 ;
```

Here, `a` is declared to be an array of size 10 at line 1. `x` is initialized to 0 at line 2 and `a[x]` is initialized to 0 at line 3.

Considering the safety of the expression `a[x] = 0` (ϕ), the set $\Psi_{sp}(\phi)$ is {`(3, (a[x] = 0)), (2, (x = 0))`}. Now, for each of these program fragments, we apply the appropriate templates for array bounds and generate explanations by combining them with the program variables `a` and `x` along with their labels. In this case, the safety explanation is:

The access `a[x]` *at line 3 is safe as the term* `x` *is evaluated from* `x = 0` *at line 2;* `x` *is within* `0` *and* `9`*; and hence the access is within the bounds of the array declared at line 1.*

Now if we were interested in initialization of variables, the set $\Psi_{sp}(\phi)$ is still {`(3, (a[x] = 0)), (2, (x = 0))`}. However, the template definitions for the same fragments differ and the explanation is:

The assignment `a[x] = 0` *at line 3 is safe; the term* `x` *is initialized from* `x=0` *at line 2.*

5 Implementation and Illustration

We now describe an implementation of the safety document generator based on the principles discussed in the previous sections and give an example of how it works for different safety policies.

5.1 Implementation

The process of generating the explanations from the intermediate language can be broadly classified into two phases.

- Labeling the intermediate code and generating verification conditions.
- Scanning the verification conditions and generating explanations.

We scan the verification conditions to identify the different parts of the program that require safety explanations collecting as much information as possible about the different data and variables along the way, and computing the $\Psi_{sp}(\phi)$. Fragments that require safety explanations differ for different safety policies. Since we analyze the verification conditions and not the program, the safety policy has already determined this. For example, in safety with respect to array bounds, the fragments that require explanations would be the array accesses in the program. On the other hand, we need to consider all variable assignments, array accesses and assignments, declarations, and conditional sentences for safety with respect to initialization of variables. That is, we consider all fragments where a variable is used and determine whether the variable has been initialized. In addition, we also accumulate information about the program variables, constants and the blocks.

Finally, using the information that we have accumulated during the scanning phase, we generate explanations for why the program is safe. As we have already mentioned, our tool is designed to be generic. Irrespective of the safety policy

that we are currently concerned with, the tool analyzes each fragment that requires an explanation, and generates explanations using templates as discussed in the previous section. It should be noted that such an approach makes extension very easy as the introduction of a new safety policy would only involve providing definitions in the domain of the safety property for each template.

5.2 A Simple Example

We give an example, here, of some intermediate code and the corresponding explanations provided by the document generator.

```
0 proc(eg)
   {
1   a[10] : int
2   b : int ;
3   c : int ;
4   d : int ;

5   b = 1 ;
6   c = 2 ;
7   d = b*b + c*c ;

8   for(i=0;i<10;i++)
     {
9     if(i < 5)
10      a[d+i] = d ;
            else
11      a[2*d-1-i] = d ;
     }
   }
```

The explanations generated for safety with respect to array bounds and initialization are given in Figures 4 and 5, respectively.

6 Design Issues

In this section, we present some issues that were analyzed during the design and implementation of the safety document generator and then describe features that we have implemented for flexibility.

6.1 Invariants

To enable the document generator to recognize those parts of the verification conditions which come from loop invariants, we need to specifically label them with labels of the form `inv`(I). Then, while generating explanations for fragments within loops, we first find if the loop has an explicit invariant. If it does, we check

Safety Explanations for Array Bounds
The access `a[d+i]` *at line* `10` *(if the condition at line* `9` *is true) is safe as the term* `d` *is evaluated from* `d=b*b+c*c` *at line* `7`*; the term* `b` *is evaluated from* `b=1` *at line* `5`*; the term* `c` *is evaluated from* `c=2` *at line* `6`*; for each value of the loop index* `i` *from* `0` *to* `9` *at line* `8`*;* `d+i` *is within* `0` *and* `9`*; and hence the access is within the bounds of the array declared at line* `1`.
The access `a[2*d-1-i]` *at line* `11` *(if the condition at line* `9` *is false) is safe as the term* `d` *is evaluated from* `d=b*b+c*c` *at line* `7`*; the term* `b` *is evaluated from* `b=1` *at line* `5`*; the term* `c` *is evaluated from* `c=2` *at line* `6`*; for each value of the loop index* `i` *from* `0` *to* `9` *at line* `8`*;* `2*d-1-i` *is within* `0` *and* `9`*; and hence the access is within the bounds of the array declared at line* `1`.

Fig. 4. Auto-generated Explanation: *array* safety policy

Safety Explanations for Initialization of Variables
The assignment `b=1` *at line* `5` *is safe.*
The assignment `c=2` *at line* `6` *is safe.*
The assignment `d=b*b+c*c` *at line* `7` *is safe; the term* `b` *is initialized from* `b=1` *at line* `5`*; the term* `c` *is initialized from* `c=2` *at line* `6`.
The loop index `i` *ranges from* `0` *to* `9` *and is initialized at line* `8`.
The conditional expression `i<5` *appears at line* `9`*; the loop index* `i` *ranges from* `0` *to* `9` *and is initialized at line* `8`.
The assignment `a[d+i]=d` *at line* `10` *is safe (if the condition at line* `9` *is true) ; the term* `d` *is initialized from* `d=b*b+c*c` *at line* `7`*; the term* `b` *is initialized from* `b=1` *at line* `5`*; the term* `c` *is initialized from* `c=2` *at line* `6`*; the loop index* `i` *ranges from* `0` *to* `9` *and is initialized at line* `8`.
The assignment `a[2*d-1-i]=d` *at line* `11` *is safe (if the condition at line* `9` *is false) ; the term* `d` *is initialized from* `d=b*b+c*c` *at line* `7`*; the term* `b` *is initialized from* `b=1` *at line* `5`*; the term* `c` *is initialized from* `c=2` *at line* `6`*; the loop index* `i` *ranges from* `0` *to* `9` *and is initialized at line* `8`.

Fig. 5. Auto-generated Explanation: *init* safety policy

if the fragment shares any variables with the invariant. The idea behind this is that it is always possible that the loop invariant might be completely unrelated to the safety of a fragment within that loop. In such a case, our explanations should not consider the loop invariant. However, if the invariant does (presumably) affect the safety of a fragment, we incorporate it into the explanation using the label giving the line at which the invariant was defined.

6.2 Two-phase approach

We use a two-phase approach where the first phase involves scanning the program and accumulating information while explanations are generated in the second phase. It could be argued that explanations could be generated on the fly, while scanning, rather than in two phases. The reason behind having a separate scanning phase from the document generation phase is to support multiple queries regarding the safety of a program. The user might want to determine the

safety of specific lines in the program and might want to do it more than once. In such a scenario, a tool would have to scan the code and accumulate information each and every time. On the other hand, our current approach ensures that the program is scanned only once even in case of multiple and/or repetitive queries.

6.3 Program Slicing

The document generator described so far analyzes each program fragment with a view of providing complete and comprehensive safety explanations. This technique combined with the def-use analysis tends to make the reports long. Moreover, users might be interested in specific parts of the program rather than the entire program. To accommodate this, we adopt the idea of a *program slice*. A program slice comprises those parts of the program that actually determine the state of a given variable at a particular point in execution. We give users the option of checking a *slice* of the program rather than the entire program. Users interested in a particular block can specify just the lines numbers within that block. It is also possible that users could be interested in a few specific variables. In such a case, they can just mention the variables involved. In both these cases, the document generator provides safety explanations only for those fragments that fall within the area of interest. However, the safety of these fragments could depend on the safety of other fragments so we still need to track each program term to its origin while generating appropriate explanations along the way.

6.4 Ranking

We have designed the document generator to be comprehensive. For some of the more complex programs synthesized by AutoBayes, the safety document can run to over a hundred pages. Although slicing can be used to focus attention on areas of interest, it is still nice to get an overall justification of why a program is safe.

Clearly, some facts are more important than others. We have implemented a simple heuristic which ranks the fragments and displays them based on user request. For instance, initialization of variables to constants can be viewed as a trivial command so the corresponding explanation can be eliminated. We have categorized fragments (in order of increasing priority) in terms of assignments to numeric constants, loop variable initializations, variable initializations, array accesses and — the highest priority — any command involving invariants.

The rationale behind giving explanations involving invariants the highest priority is that invariants are generally used to fill in the trickiest parts of a proof, so are most likely to be of interest.

7 Conclusions and Future Work

The documentation generation system which we have described here builds on our state-of-the-art program synthesis system, and offers a novel combination

of synthesis, verification and documentation. We believe that documentation capabilities such as this are essential for formal techniques to gain acceptance.

Our plan is to combine the safety documentation with ongoing work on *design documentation*. We currently have a system which is able to document the synthesized code (explaining the specification, design choices made during synthesis, and so on), either as in-line comments in the code or as a browsable document, but it remains to integrate this with the safety document generator. We intend to let the user chose between various standard formats for the documentation (such as those mandated by DO-178B or internal NASA requirements).

A big problem for NASA is the recertification of modified code. In fact, this can be a limiting factor in whether a code change is feasible or not. For synthesis, the problem is that there is currently no easy way to combine manual modifications to synthesized code with later runs of the synthesis system. We would like to be able to generate documentation which is specific to the changes which have been made.

Finally, we intend to extend our certification system with new policies (including resource usage, and constraints on the implementation environment). The two safety policies which we have illustrated this with here are *language-specific* in the sense that the notion of safety is at the level of individual commands in the language. We have also looked at *domain-specific* policies (such as for various matrix properties) where the reasoning takes place at the level of code blocks. This will entail an interesting extension to the document generator, making use of domain-specific concepts.

References

[BBC+97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual: Version 6.1. Technical Report RT-0203, 1997.

[BRLP98] Jeffrey Van Baalen, Peter Robinson, Michael Lowry, and Thomas Pressburger. Explaining synthesized software. In D. F. Redmiles and B. Nuseibeh, editors, *Proc 13th IEEE Conference on Automated Software Engineering*, pages 240–248, 1998.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[CKT95] Y. Coscoy, G. Kahn, and L. Théry. Extracting text from proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proc. Second International Conference on Typed Lambda Calculi and Applications, Edinburgh, Scotland*, volume 902, pages 109–123, 1995.

[DF03] Ewen Denney and Bernd Fischer. Correctness of source-level safety policies. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Proceedings FM 2003: Formal Methods*, volume 2805 of *Lect. Notes Comp. Sci.*, pages 894–913, Pisa, Italy, September 2003. Springer.

[DKT93] Arie Van Deursen, Paul Klint, and Frank Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5/6):523–545, 1993.

[FS03] Bernd Fischer and Johann Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *J. Functional Programming*, 13(3):483–508, May 2003.

[HMBC99] Amanda M. Holland-Minkley, Regina Barzilay, and Robert L. Constable. Verbalization of high-level formal proofs. In *AAAI/IAAI*, pages 277–284, 1999.

[Hua94] Xiaorong Huang. Proverb: A system explaining machine-found proofs. In Ashwin Ram and Kurt Eiselt, editors, *Proc. 16th Annual Conference of the Cognitive Science Society, Atlanta, USA*, pages 427–432. Lawrence Erlbaum Associates, 1994.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[Pol] PolySpace Technologies. `http://www.polyspace.com`.

[WBS+01] Jon Whittle, Jeffrey Van Baalen, Johann Schumann, Peter Robinson, Thomas Pressburger, John Penix, Phil Oh, Mike Lowry, and Guillaume Brat. Amphion/NAV: Deductive synthesis of state estimation software. In *Proc IEEE Conference on Automated Software Engineering*, 2001.

[WS03] Jon Whittle and Johann Schumann. Automating the implementation of Kalman filter algorithms, 2003. In review.